

## PYCLAW: ACCESSIBLE, EXTENSIBLE, SCALABLE TOOLS FOR WAVE PROPAGATION PROBLEMS

DAVID I. KETCHESON\*, KYLE T. MANDLI<sup>†</sup>, ARON J. AHMADIA<sup>‡</sup>, AMAL ALGHAMDI<sup>§</sup>, MANUEL QUEZADA DE LUNA<sup>¶</sup>, MATTEO PARSANI<sup>||</sup>, MATTHEW G. KNEPLEY<sup>\*\*</sup>, AND MATTHEW EMMETT<sup>††</sup>

**Abstract.** Development of scientific software involves tradeoffs between ease of use, generality, and performance. We describe the design of a general hyperbolic PDE solver that can be operated with the convenience of MATLAB yet achieves efficiency near that of hand-coded Fortran and scales to the largest supercomputers. This is achieved by using Python for most of the code while employing automatically-wrapped Fortran kernels for computationally intensive routines, and using Python bindings to interface with a parallel computing library and other numerical packages. The software described here is PyClaw, a Python-based structured grid solver for general systems of hyperbolic PDEs [18]. PyClaw provides a powerful and intuitive interface to the algorithms of the existing Fortran codes Clawpack and SharpClaw, simplifying code development and use while providing massive parallelism and scalable solvers via the PETSc library. The package is further augmented by use of PyWENO for generation of efficient high-order weighted essentially non-oscillatory reconstruction code. The simplicity, capability, and performance of this approach are demonstrated through application to example problems in shallow water flow, compressible flow and elasticity.

**1. Introduction.** Traditionally, scientific codes have been developed in compiled languages like Fortran or C. There exists an abundance of well-tested, efficient, often serial implementations of numerical algorithms in those languages. It is often desirable to parallelize and extend such codes with new algorithmic components in order to apply them to increasingly challenging problems.

More recently, high-level scientific programming languages, such as MATLAB, IDL, and R, have also become an important platform for numerical codes. These languages offer powerful advantages: they allow code to be written in a language more familiar to scientists and they permit development to occur in an evolutionary fashion. Problem parameters can be specified and plotting can be performed interactively, bypassing the comparatively slow edit/compile/run/plot cycle of development in Fortran or C [23]. However, programs written in such high-level languages are not portable to high performance computing platforms and may be very slow compared to compiled code.

We present one approach to leveraging the advantages of both kinds of languages. Our starting point is Clawpack [16]: a widely used, state-of-the-art package for solving hyperbolic systems of partial differential equations, such as those arising in fluid mechanics, astrophysics, geodynamics, magnetohydrodynamics, oceanography, porous me-

---

\*King Abdullah University of Science and Technology, Box 4700, Thuwal, Saudi Arabia, 23955-6900 (david.ketcheson@kaust.edu.sa)

<sup>†</sup>University of Texas at Austin, 1 University Station C0200 Austin, TX 78712-0027 (kyle@ices.utexas.edu)

<sup>‡</sup>King Abdullah University of Science and Technology, Box 4700, Thuwal, Saudi Arabia, 23955-6900 (aron.ahmadia@kaust.edu.sa)

<sup>§</sup>King Abdullah University of Science and Technology, Box 4700, Thuwal, Saudi Arabia, 23955-6900 (amal.alghamdi@kaust.edu.sa)

<sup>¶</sup>Texas A&M University, College Station, TX 77843 (mquezada@math.tamu.edu)

<sup>||</sup>King Abdullah University of Science and Technology, Box 4700, Thuwal, Saudi Arabia, 23955-6900 (matteo.parsani@kaust.edu.sa)

<sup>\*\*</sup>University of Chicago, 5735 S. Ellis Ave. Chicago, IL 60637 (knepley@ci.uchicago.edu)

<sup>††</sup>University of North Carolina at Chapel Hill, Chapel Hill, NC 27599 (memmett@unc.edu)

dia flow, and numerous other fields. In this paper we present PyClaw, a package that brings greater accessibility, flexibility, and parallelization to Clawpack and related packages. PyClaw is used as an illustration to describe, demonstrate, and provide support for a particular approach to construction of scientific software. The approach we advocate consists of three steps:

- (i) use Python to create a convenient interface to serial legacy code;
- (ii) use Python to interface the resulting code with software tools for parallelization of the code, with minimal modification of the serial legacy code;
- (iii) use Python to interface the resulting parallel code to other packages that provide extended functionality.

In the case of PyClaw, (i) consists of a Python interface to the Clawpack and SharpClaw Fortran-based packages for numerical solution of systems of hyperbolic PDEs. This interface allows the code to be operated in the same convenient and interactive way that one works with MATLAB. In step (ii), PyClaw was parallelized by interfacing with PETSc, a state-of-the-art library for parallel scientific computing. This enables parallel computation that scales to large supercomputers and achieves on-core performance close to that of the legacy code. Finally, step (iii) is illustrated in that PyClaw was interfaced with PyWENO, to increase the available order of accuracy of numerical approximation. A key in all three steps is the use of the numerical Python package numpy [22]. The idea of using a layer of numpy-based Python code on top of Fortran, C, or C++ kernels to solve PDEs has become increasingly popular over the past several years; see for instance [19, 6]. We consider PyClaw to be a very convincing case study.

PyClaw is one of the most highly scalable Python-based codes available. Perhaps the first well-known scientific project to provide a parallel solver in Python is GPAW, which extends the Python interpreter itself with parallel data structures and algorithms for electronic structure calculations [20]. Python has also previously been used as a tool for parallelizing Fortran and C codes by introducing parallel code in a Python layer that also calls the Fortran/C kernels [21]. In the FiPy package, parallelism is achieved by using an existing parallel library (Trilinos) through its Python bindings [6]. PyClaw takes an approach similar to that of FiPy, in which all parallel operations over distributed-memory processes are handled by the PETSc library through the petsc4py Python package (<http://code.google.com/p/petsc4py/>). This approach offers the advantages of utilizing a documented, well-designed abstract parallel interface for developers that is already known to achieve excellent scaling on many architectures.

The algorithms of Clawpack and its high-order extension, SharpClaw, are described in Section 2. The PyClaw framework is described in Section 3. Section 4 describes the parallelization of PyClaw. As demonstrated in Section 5, PyClaw maintains both the serial performance of Clawpack and the parallel scalability of PETSc. In Section 6, we briefly highlight some of the software development practices that have contributed to the success of PyClaw. The combination of wave propagation algorithms and scalable parallelization enables efficient solution of interesting scientific problems, as demonstrated through three examples in Section 7.

A repository containing the data, software, and hardware environments for reproducing all experimental results and figures in this paper is available online (<http://>

Listing 2: Solution of a 2D Euler Riemann problem

```
1 from clawpack import pyclaw
2 from clawpack import riemann
3 # Define the equations and the solution algorithm
4 solver = pyclaw.ClawSolver2D(riemann.rp2_euler_4wave)
5 # Define the problem domain
6 domain = pyclaw.Domain([0.,0.],[1.,1.],[100,100])
7 solver.all_bcs = pyclaw.BC.extrap
8 solution = pyclaw.Solution(solver.num_eqn,domain)
9 # Set physical parameters
10 gamma = 1.4
11 solution.problem_data['gamma'] = gamma
12 # Set initial data
13 xx,yy = domain.grid.p_centers
14 l = xx<0.5; r = xx>=0.5; b = yy<0.5; t = yy>=0.5
15 solution.q[0,...] = 2.*l*t + 1.*l*b + 1.*r*t + 3.*r*b
16 solution.q[1,...] = 0.75*t - 0.75*b
17 solution.q[2,...] = 0.5*l - 0.5*r
18 solution.q[3,...] = 0.5*solution.q[0,...]* \
19                 (solution.q[1,...]**2+solution.q[2,...]**2) \
20                 + 1./(gamma-1.)
21 # Solve
22 solver.evolve_to_time(solution,tend=0.3)
23 # Plot
24 pyclaw.plot.interactive_plot()
```

[bitbucket.org/ahmadi/pyclaw-sisc-rr](http://bitbucket.org/ahmadi/pyclaw-sisc-rr)). The most recent release of the PyClaw code is hosted at <http://github.com/clawpack/pyclaw> and can be installed alongside its dependencies in the clawpack distribution in a few minutes with the following pip commands (pip is a freely available Python package installer and manager):

Listing 1: Installing the most recent release of PyClaw and its dependencies

```
1 pip install numpy
2 pip install clawpack
```

The `petsc4py` package is not an explicit dependency of PyClaw, but if it has been installed, the PetClaw parallel extension described in Section 4 is seamlessly enabled.

It is our hope that readers will download and try the code. To whet the reader's appetite, an example of a complete PyClaw program is shown in Listing 2. This example sets up, runs, and plots the solution of a two-dimensional inviscid fluid dynamics problem (specifically, test case 6 of [13]).

**2. Finite Volume Hyperbolic PDE solvers.** The numerical methods in PyClaw compute approximate solutions of systems of hyperbolic conservation laws:

$$\kappa(\mathbf{x})\mathbf{q}_t + \nabla \cdot \mathbf{f}(\mathbf{q}, \mathbf{x})_x = \mathbf{s}(\mathbf{q}, \mathbf{x}). \quad (2.1)$$

Here  $\mathbf{q}(\mathbf{x}, t)$  is a vector of conserved quantities (e.g., density, momentum, energy) and  $\mathbf{f}(\mathbf{q}, \mathbf{x})$  represents the flux (modeling wave-like phenomena), while  $\mathbf{s}(\mathbf{q}, \mathbf{x})$  represents additional non-hyperbolic *source* terms, such as diffusion or chemical reactions. The *capacity function*  $\kappa(\mathbf{x})$  is frequently useful for taking into account variations in material properties or in the use of non-uniform grids (see [15, Chapter 6]). Here we describe high-resolution shock capturing methods, which is one of the most successful classes of numerical methods for solving (2.1).

Computing solutions to nonlinear hyperbolic equations is often costly. Solutions of (2.1) generically develop singularities (shocks) in finite time, even if the initial data are smooth. Accurate modeling of solutions with shocks or strong convective character requires computationally expensive techniques, such as Riemann solvers and nonlinear limiters.

In a finite volume method, the unknowns at time level  $t^n$  are taken to be the averages of  $q$  over each cell:

$$Q_i^n = \frac{1}{\Delta x} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} q(x, t^n) dx, \quad (2.2)$$

where  $\Delta x = x_{i+\frac{1}{2}} - x_{i-\frac{1}{2}}$  and  $i$  are the local grid spacing and the cell's index, respectively. A simple update of the cell averages based on the resulting waves gives the classic Godunov method, a robust but only first-order accurate numerical scheme:

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x} \left( F_{i+\frac{1}{2}}^n - F_{i-\frac{1}{2}}^n \right), \quad (2.3)$$

where  $F$  and  $\Delta t$  are the *numerical flux* function and the time step. Godunov's method results from taking a particular choice of  $F$  referred to as the *upwind flux*.

The first-order method just described is very dissipative. Higher-order extensions require the computation of higher derivatives of the solution or flux. Near a solution discontinuity, or shock, spurious oscillations tend to arise due to dispersion and numerical differencing across the discontinuity. In order to combat this, shock-capturing methods use special nonlinear algorithms to compute numerical derivatives in a non-oscillatory way by limiting the value of the computed derivative in the vicinity of a discontinuity [15].

The classic Clawpack algorithm is based on the second-order Lax-Wendroff difference scheme that was later extended by LeVeque [14, 15]. This scheme can be written in the flux-differencing form (2.3) by an appropriate choice of numerical flux, which has the form

$$F_{i-\frac{1}{2}}^n = F_{\text{upwind}} + F_{\text{correction}}, \quad (2.4)$$

where  $F_{\text{upwind}}$  is the Godunov flux. The classic Clawpack algorithm is based on modifying (2.4) by applying a limiter to  $F_{\text{correction}}$ . However, in LeVeque’s extension, rather than applying the limiter to the flux variables, the limiter is applied directly to the waves computed by the Riemann solver. This allows for better accuracy by limiting only the characteristic families that are discontinuous in a given neighborhood. Furthermore, the first-order contribution is written in terms of *fluctuations* (which approximate the quasi-linear term  $Aq_x$ ) rather than fluxes (which approximate  $f(q)$ ). This allows the algorithm to be applied to hyperbolic systems not in conservation form.

While the Lax-Wendroff approach can be extended to even higher order, this is cumbersome because of the large number of high order terms appearing in the Taylor series. A simpler alternative is the method of lines, in which the spatial derivatives are discretized first, leading to a system of ODEs that can be solved by traditional methods. This is the approach taken in SharpClaw [10, 9, 11]. First, a non-oscillatory approximation of the solution is reconstructed from the cell averages to give high order accurate point values just to the left and right of each cell interface. This reconstruction is performed using weighted essentially non-oscillatory (WENO) reconstruction in order to avoid spurious oscillations near discontinuities. As in the classic Clawpack algorithm, the scheme is written in terms of fluctuations rather than fluxes, so that it can be applied to non-conservative problems.

**2.1. Clawpack and SharpClaw.** We now describe the "legacy" Fortran codes on which PyClaw is built.

The classic algorithm implemented in Clawpack (“Conservation Laws Package”) includes extensions to two and three dimensions, adaptive mesh refinement, and other enhancements [16]. The unsplit multidimensional Clawpack algorithms include additional correction terms, computed by a secondary “transverse” Riemann solver, which approximates corner transport terms. The Clawpack software (<http://www.clawpack.org>) and its extensions, consisting of open source Fortran code, have been freely available since 1994. More than 7,000 users have registered to download Clawpack.

Clawpack is a very general tool in the sense that it is easily adapted to solve any hyperbolic system of conservation laws. The only specialized code required in order to solve a particular hyperbolic system is the Riemann solver routine. A wide range of Riemann solvers, including several for the most widely studied hyperbolic systems, have been developed by Clawpack users and are also freely available. Clawpack handles not only simple Cartesian grids but any logically quadrilateral grid, provided that there is a map from a uniform Cartesian domain to the desired physical domain (see Section 7.1). Non-hyperbolic source terms ( $\mathbf{s}(\mathbf{q}, \mathbf{x})$ ) can be easily included via operator splitting. For more examples and details regarding Clawpack, see [14] and [15, Chapter 23].

The high-order WENO-based wave propagation method is implemented in SharpClaw, another Fortran package designed similarly to Clawpack and which makes use of the same Riemann solver routines. The default options in SharpClaw employ fifth-order WENO reconstruction in space and the fourth-order strong stability preserving (SSP) Runge–Kutta method of [8] in time. In multi-dimensions SharpClaw requires propagation of waves only in the normal direction to each edge.

**3. PyClaw.** PyClaw is an object-oriented framework that incorporates the functionality of Clawpack and SharpClaw. This functionality may be provided via either pure Python or calls made to the underlying Fortran routines included in Clawpack and SharpClaw. PyClaw is designed such that the user provides appropriate call-back functions, such as a Riemann solver, that describe and implement the problem in question, PyClaw then manages the “main” routine including appropriate time stepping and output. It also avoids the need to deal with strictly formatted data files and reduces the need to write custom Fortran routines for new problems. Instead, problems can be set up interactively or in a simple scripting language. PyClaw also allows for simulation and visualization to be done in a single, interactive environment. Users may engage with this framework at different levels, depending on their expertise and requirements. These interfaces are described in Section 3.1.

PyClaw wraps the full functionality of the “classic” 1D, 2D and 3D Clawpack code including the use of capacity functions, mapped grids, and both dimensionally-split and fully-multidimensional algorithms. It also provides the full functionality of SharpClaw, and adds to this with higher order WENO reconstruction. It does not presently include adaptive mesh refinement, which is part of the AMRClaw and GeoClaw extensions of Clawpack.

**3.1. Interfaces.** The PyClaw distribution includes pre-written application scripts that solve problems in acoustics, elasticity, compressible flow, shallow water flow, and other application domains. These application scripts represent the typical “main” routines that lower-level language developers are used to, and are written with ease of understanding as their most important goal. These scripts run both on serial workstations and from batch processing queues for, e.g., 8,000-node parallel jobs without modification. Novice users can solve a wide range of problems by modifying the example scripts to deal with different domains, initial conditions, boundary conditions, and so forth. This requires only simple understanding of high-level scripting code, but allows users to compute solutions of complex problems on large supercomputers. The scripts for the applications in this paper have all been added to the distributed examples, and we plan to continue this practice with respect to future publications that use PyClaw.

Novice or advanced users may also run problems and analyze results in an interactive shell. When Python is invoked with standard input connected to a tty device, it reads and executes commands interactively. This feature simplifies serial development, debugging, and visualization, and is familiar to users of commercial software such as MATLAB and Mathematica. PyClaw’s top level classes present the same API whether used in serial or parallel, allowing users to develop interactively, then run production jobs in batch environments.

Advanced users may want to solve a hyperbolic system that is not included among the example applications. In Clawpack a number of Riemann solvers have been written for specific systems and are included with Clawpack. Since PyClaw and Clawpack Riemann solvers are interoperable, many systems of interest have already been implemented and can be used immediately in PyClaw. A user also has the option of writing his/her own Riemann solver in Fortran which can then be utilized in Clawpack as well as PyClaw.

Clawpack users who wish to run an existing serial Clawpack application in parallel can do so easily by wrapping any problem-specific Fortran routines (such as those that set initial conditions or compute source terms) automatically with `f2py` and using the resulting Python function handles in PyClaw.

Numerical analysts are often interested in comparing solutions of a problem obtained with different methods or different values of method parameters. The sample application scripts provide a common functional interface that can be accessed from the command line for selecting between solvers, choosing serial or parallel computation, and other options. Users are free to extend this interface to allow more programmatic flexibility at the command line or from within a batch environment. PyClaw also fully exposes the full range of command line options available from the underlying PETSc library, allowing advanced users to tweak low-level settings such as message-passing communication strategies.

Frequently, research scientists are interested in comparing the performance of numerical methods. PyClaw enables this comparison by allowing scientific developers to extend the software with a new Solver class. The Solver class, described in the next section, is responsible for prescribing a single time step of the numerical algorithm. In PyClaw this is accomplished by implementing a `homogenous_step` routine which evolves the solution of the PDE from time  $t$  to  $t + \Delta t$ . Since most existing codes have such a routine already, it is often straightforward to include legacy code in the PyClaw framework by simply wrapping this function. Non-hyperbolic terms  $\mathbf{s}(\mathbf{q}, \mathbf{x})$  can also be incorporated via operator splitting. This is the most common way to extend the Solver class and allows for easy comparison between different numerical methods.

**3.2. Classes.** The primary abstractions used in PyClaw are the Solver and the Solution. The Solver class provides an abstract interface to evolve a Solution object forward in time. The Solution class is a data abstraction containing information about the domain of the PDE and the state of the solution at a particular time inside the domain. Here we will discuss these classes and how they interact.

The role of the Solver is illustrated in Figure 3.1(b). The Solver class prescribes how a State  $Q^n$  is evolved forward in time to obtain  $Q^{n+1}$ ; in general this consists of three parts:

- (i) set ghost cell values based on the prescribed boundary conditions;
- (ii) advance the solution based on the hyperbolic terms (i.e.,  $\mathbf{q}_t + \nabla \cdot \mathbf{f}(\mathbf{q}, \mathbf{x}) = 0$ );
- (iii) advance the solution based on the source term  $\mathbf{s}(\mathbf{q}, \mathbf{x})$  (if present) by a fractional-step approach [15, Chapter 17].

The base Solver class implements the basic interface to each of these functions and a subclass of the Solver class is expected to implement the appropriate functions depending on the numerical method being implemented. The Solver class is sufficiently abstract to accommodate algorithms that are based on the method of lines, such as SharpClaw, as well as algorithms that are not, such as Clawpack.

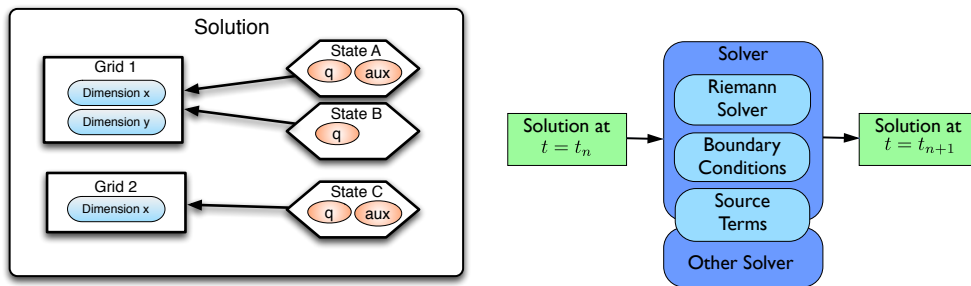
The Solution class, depicted in Figure 3.1(a), has two purposes:

- Describe the problem domain
- Keep track of the values of the state variables  $Q^n$  and PDE coefficients

Like Clawpack, PyClaw simulations are always based on a computational domain com-

posed of tensor products of one-dimensional equispaced discretizations of space. More general physical domains may be used as long as it is possible to map them to that computational domain. PyClaw includes a set of geometry classes that implement these abstractions.

The solution values  $Q^n$  and the PDE coefficients are contained in numpy arrays stored in the Solution object. Thus the full Solution class represents a snapshot of the gridded data. The class acts as a container object with one or more Grid and State objects such as in the case of adaptive mesh refinement or nested grids, both of which are possible with Clawpack algorithms, though not yet available in PyClaw. This hierarchical class structure allows the underlying data and algorithms to be modified without the knowledge of the interacting objects and without changing the interface presented to the user. An example of this is the PetClaw State object, which reimplements State functionality over distributed memory. In the future, we intend to provide State objects that implement other strategies for accelerating performance.



(a) Structure of a PyClaw Solution object, which may contain multiple Grid objects, each of which may have multiple associated State objects. Each State object has a associated fields of conserved quantities (e.g., density, momentum) and optionally, associated auxiliary property fields.

(b) Role of Solver object. The Solver acts on a Solution object in order to evolve it to a later time.

Fig. 3.1: Structure and function of the main PyClaw classes.

**3.3. Extension using PyWENO.** One of the principal achievements of PyClaw has been to facilitate the extension of Clawpack and SharpClaw by interfacing with other packages. For example, PyWENO [3] has been used to add much higher-order functionality to the existing SharpClaw code, within PyClaw.

The Fortran code SharpClaw contains only fifth-order WENO routines for evaluation at cell interfaces. New WENO routines for PyClaw were generated by PyWENO, which is a standalone package for building custom WENO codes. For a given (arbitrarily high) order of reconstruction, PyWENO symbolically computes the smoothness coefficients, reconstruction coefficients, and optimal (linear) weights of the WENO method. From these symbolic results, PyWENO generates Fortran kernels that perform the WENO reconstructions (it can also generate C and OpenCL kernels). The generated kernels can optionally perform individual steps of the reconstruction process (i.e., computing smoothness indicators, nonlinear weights, or the final reconstruction) or combinations thereof. This



affords authors some flexibility in avoiding redundant computations or minimizing memory allocations and accesses. Furthermore, the points within each cell at which the WENO reconstruction is performed are also arbitrary. Negative weights are automatically split into positive and negative parts [26], allowing PyWENO to generate kernels for routines for arbitrary sets of points (such as arbitrary order Gauss-Legendre quadrature points).

For PyClaw, odd-order WENO routines to approximate the solution at the left and right edges of each cell were generated from fifth to seventeenth order. All aspects of the WENO reconstruction are wrapped into standalone subroutines and no temporary work arrays are allocated. Using these routines is trivially easy; for instance, to use the 9th-order WENO method instead of the classic algorithm in Listing 2 one needs only to replace line 9 by the two lines

Listing 3: Using SharpClaw (with PyWENO)

```
1 solver = pyclaw.SharpClawSolver1D()
2 solver.weno_order = 9
```

**4. Parallelization.** Like many finite difference and finite volume codes, Clawpack and SharpClaw implement boundary conditions through the use of *ghost cells*. In this approach, fictitious layers of cells are added around the edge of the problem domain; the number of layers depends on the width of the stencil of the numerical scheme. At the beginning of each step, the ghost cell values are set to satisfy the specified boundary conditions. Then the numerical scheme is applied on all the interior (non-ghost) cells. Many types of boundary conditions can be handled well in this manner, including periodicity, reflection, and outflow (non-reflecting). Custom boundary conditions may also be specified, for instance to model time-dependent inflow.

This approach is highly amenable to parallelization, since it is based on the idea that information at the edge of a domain is filled in by a routine that is independent of the rest of the numerical scheme. Therefore, the serial kernels can be applied on each processor of a distributed parallel machine as long as some routine first fills the ghost cells on the processor either by appeal to boundary conditions or through communication with neighboring processors, as appropriate. Only this ghost cell routine needs to know the global topology of the problem; the serial kernels operate based entirely on local information. This orthogonality allows independent development of serial numerical schemes and parallel communication patterns, and is a key strategy in combining the work of computational mathematicians and computer scientists.

The same global-local decomposition is employed in PETSc. The PETSc library includes a DMDA object that implements parallelization through the use of ghost cells. The DMDA is a highly scalable class for data layout across parallel, structured grids. All storage allocation and communication of ghost values is handled by the DMDA, but storage is returned to the PyClaw program as numpy arrays so that no code changes are necessary and the user experience is identical. In Figure 4.1, we show three different representations of data over a simple  $5 \times 6$  structured grid. The global ordering is used as input to PETSc linear and nonlinear solvers, whereas the natural ordering is used for output since it is independent of the particular parallel partition. Local ordering is used to extract data over

Proc 2			Proc 3	
25	26	27	28	29
20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4
Proc 0			Proc 1	

Natural numbering

Proc 2			Proc 3	
21	22	23	28	29
18	19	20	26	27
15	16	17	24	25
6	7	8	13	14
3	4	5	11	12
0	1	2	9	10
Proc 0			Proc 1	

Global numbering

Proc 2			Proc 3	
X	X	X	X	X
X	X	X	X	X
12	13	14	15	X
8	9	10	11	X
4	5	6	7	X
0	1	2	3	X
Proc 0			Proc 1	

Local numbering

Fig. 4.1: A simple  $5 \times 6$  structured mesh is represented using a DMDA. The leftmost figure shows the natural numbering of unknowns, for instance used by most visualization packages, which is independent of the parallel layout. The middle figure shows the PETSc global numbering of unknowns, which is contiguous for each process. The rightmost figure shows the local numbering of unknowns for process 0, which includes ghost unknowns shared with neighboring processes.

a “halo” region, including ghost unknowns shared with other processes.

This is, in fact, how PyClaw makes use of the DMDA structure. Local vectors are extracted with a given number of overlap unknowns, and computations are performed using the same serial routines. These local vectors are then used to update a global vector, and PETSc performs the appropriate accumulation for shared unknowns. This simple mechanism in PETSc for integrating local and global data (which works also for unstructured grids) allows easy parallelization. Thus PyClaw relies on Clawpack and SharpClaw to provide computational kernels for time-dependent nonlinear wave propagation and on PETSc (through `petsc4py`) to manage distributed data arrays and the communication between them. The data structures in PETSc and Clawpack/SharpClaw are directly interfaced through the Python package `numpy` [22].

The parallel extension of PyClaw consists of only about 300 lines of Python code. Any PyClaw script can be run in parallel simply by replacing the statement `from clawpack import` with

```
Listing 4: Running in parallel
1 from clawpack import petclaw as pyclaw
```

and invoking the Python script with `mpirun`.

The serial PyClaw routines handle discretization, Riemann solves, limiting and reconstruction, since they only depend on local data. PETSc handles parallel layout and communication, but has no information about the local computations. PETSc allows fine-grained control of the ghost value communication patterns so that parallel performance can be tuned to different supercomputing architectures, but by default a user does not need to manage parallelism or see PETSc code. In fact, the PetClaw user is shielded from PETSc in much the same way that a PETSc user is shielded from MPI. This separation can enable future development. For instance, an unstructured mesh topology of hexahedral

elements could be managed by PETSc, using a Riemann solver which could accommodate deformed elements, without changes to PyClaw.

In addition to communication of ghost cell values, parallel hyperbolic solvers require communication of the maximum wave speed occurring on each processor in order to check whether a prescribed stability condition (generally phrased in terms of the Courant number) has been satisfied and choose the size of the next time step appropriately. This is also handled by a single PETSc call.

Although the basic Clawpack and SharpClaw algorithms are explicit and require no algebraic solver, a powerful advantage gained by using PETSc for parallelization is the possibility of employing PETSc's solvers for implicit integration of hyperbolic problems that are stiff due to source terms or fast waves that do not need to be accurately resolved. This is the subject of ongoing work.

A particular challenge of using Python is that most parallel debuggers support only C or Fortran, making parallel debugging of Python codes difficult [4]. This is yet another motivation for using a tested parallel library like PETSc.

**5. Performance.** A few previous works have considered efficiency of scientific Python codes in serial as well as in parallel; see for instance [1, 12, 21, 4]. Those studies consisted mainly of simple code snippets run in serial or on up to a few dozen processors until the recent work [4], which includes scalability studies up to 16,384 cores. In this section, we investigate the efficiency of a full object-oriented Python framework (PyClaw) compared with hand-coded Fortran (Clawpack). We also consider the scaling of PetClaw on all 65,536 cores of the Shaheen supercomputer at KAUST.

We consider only the second-order classic Clawpack algorithm here, as we are mainly interested in the effect of using a Python framework (in the serial case) and the cost of communication (in the parallel case). In terms of these factors, roughly similar results may be expected for the performance of the higher order algorithms, and preliminary tests (not described here) indicate good scaling of those also.

**5.1. Serial performance.** For a detailed serial performance comparison of an explicit stencil-based PDE code in Python, see [12]. In that work, vectorized numpy code was found to be fast enough for some operations, while wrapped Fortran loops performed identically to a pure Fortran code. In contrast to the simple kernel code considered there, we present tests of a full object-oriented solver framework. Our results thus extend those of [12], providing an indication of the efficiency that can be expected for a sophisticated Python-based PDE solver framework.

Table 5.1 shows an on-core serial comparison between the Fortran-only Clawpack code and the corresponding hybrid PyClaw implementation for two systems of equations on two different platforms. The hyperbolic systems considered are the 2D linear acoustics equation and the 2D shallow water (SW) equations [15]. The acoustics test involves a very simple Riemann solver (amounting to a  $3 \times 3$  matrix-vector multiply) and is intended to provide an upper bound on the performance loss arising from the Python code overhead. The shallow water test involves a more typical, costly Riemann solver (specifically, a Roe solver with an entropy fix) and should be considered as more representative of realistic nonlinear application problems. Clawpack and PyClaw rely on similar Fortran kernels

that differ only in the array layout. Because most of the computational cost is in executing the low-level Fortran kernels, the difference in performance is relatively small – though not negligible. The results for the Shallow water equations are in rough agreement with the 10% overhead reported in [4]. A 10-30% increase in computational time (for realistic applications) seems well worth the advantages provided by the use of Python (in particular, easy parallelism). The overhead is expected to be even smaller for more complex systems or in three dimensions.

Table 5.1: Timing results (in seconds) for on-core serial experiments solving acoustics and shallow water problems implemented in both Clawpack and PyClaw on Intel Xeon and the IBM BlueGene/P PowerPC 450 processors

Application	Processor	Clawpack	PyClaw	Ratio
Acoustics	Intel Xeon	28s	41s	1.5
	PowerPC 450	192s	316s	1.6
Shallow Water	Intel Xeon	79s	99s	1.3
	PowerPC 450	714s	800s	1.1

**5.2. Parallel performance.** We now investigate the parallel performance of PetClaw on the Shaheen supercomputer at KAUST, an IBM BlueGene/P system consisting of 65,536 cores. When characterizing the performance of scientific codes on supercomputers, a commonly used characterization is that of *weak scalability*, which is assessed by studying how the run time of the code is affected when the resolution of the simulation and the number of processors is increased commensurately to maintain a fixed amount of work per processor. The parallel efficiency is given by dividing the run time of the single processor job by the run time of the parallel job.

The problem used for the comparisons is of a compressible, inviscid flow that consists of a shock impacting a low-density bubble (examined in detail in Section 7.2). We investigate weak scaling by running the problem for a fixed number of time steps and with a fixed number of grid cells ( $400 \times 400 = 160,000$ ) per core, while increasing the number of cores from one up to the whole machine. Figure 5.1 shows the results, with parallel efficiency provided in the last row. It is important to note that the time required to load the necessary Python packages and shared objects, which occurs only once at the beginning of a simulation (or series of batch simulations) has been excluded from the results presented here. This load time is discussed in the next section.

Observe that in all parallel runs, more than 90% of the time is spent in the computational kernels. The parallel operations scale extremely well: the the CFL condition-related reduction takes essentially the same amount of time for all runs from 16 processors up, as does the communication of ghost cell values in localToGlobal. Together these parallel operations consume about 6% of the total run time. Parallel initialization consists of PETSc parallel object construction, including memory allocation and MPI communicator initialization. Note that the parallel initialization, while significant in these artificial test runs, will not contribute significantly to the cost of real simulations because it is a one-time cost.

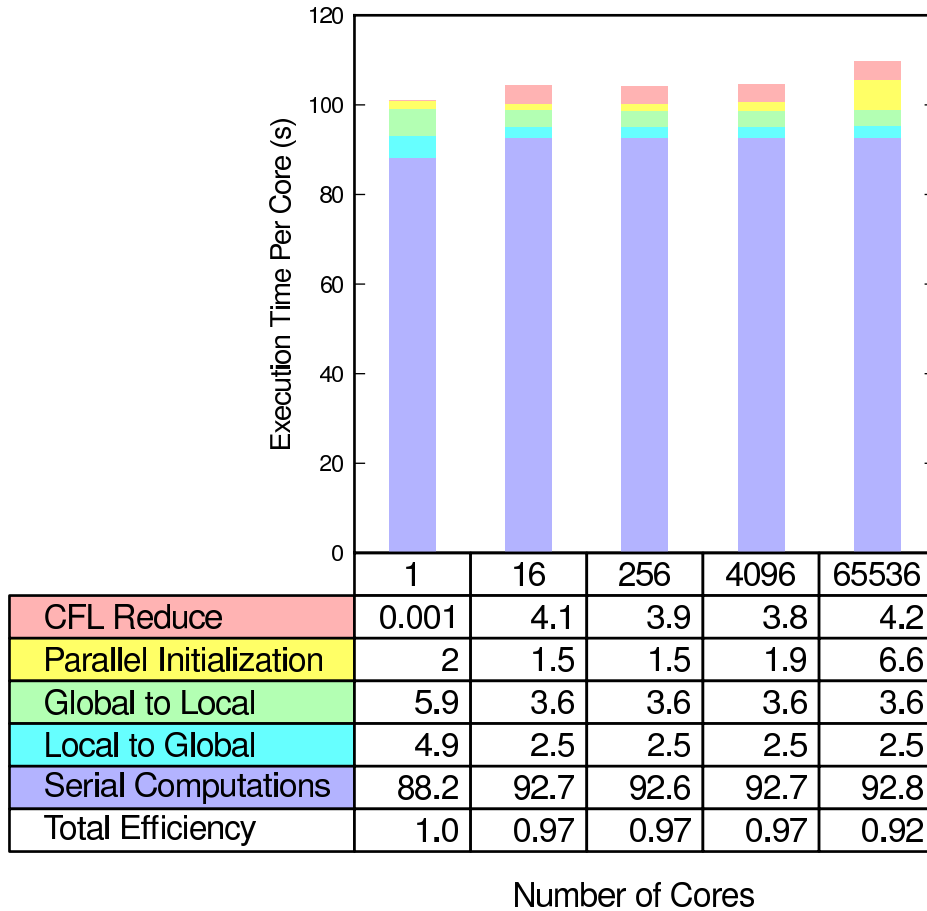


Fig. 5.1: Weak scaling performance profile of the shock bubble problem with 160,000 grid cells per core

**5.3. Dynamic Loading.** As alluded to already, the work of loading Python libraries and objects dynamically at run-time does not currently scale well on the Shaheen system. Large-scale supercomputers such as Shaheen rely on parallel file systems that are designed to support large distributed loads, with each process independently accessing data. Dynamic loading does not follow this pattern because every process is attempting to access the same data simultaneously. This issue was partially addressed in [4], but an implementation capable of supporting dynamic library loading is still lacking.

The dynamic loading time for the PetClaw runs in Section 7 is less than 5% of the total simulation time, and this will generally be the case for 2D wave propagation problems because the CFL condition means that large simulations of hyperbolic problems necessarily require long run times in order for waves to propagate across the full domain.

**6. Software Development Practices.** The majority of software development practices utilized in PyClaw are inherited from the open source software community. The community’s atmosphere of free and open sharing complements the tradition of scientific inquiry. In fact, growing movements within the scientific community seek to embrace *scientific reproducibility* for software tools used in conducting mathematical and scientific research [5, 27].

In addition to making our results reproducible, we also intend that our software be useful as a reference for understanding numerical methods involved in solving hyperbolic PDEs and as a platform for extending and applying these techniques. As such, we also seek to provide a friendly and inviting context for scientists working in this cross-disciplinary environment to conduct their research.

**6.1. Application Tests.** The goal of reproducibility in research is to improve not only confidence in results, but their extensibility as well. A series of regression tests have been devised for every major application where PyClaw has been used. The script and parameters for generating the test are stored in the repository (typically in a single file), along with verified output from a successful run. Where possible, the output is verified against a different solver implementation or by analysis. These “application tests” produce the same output regardless of the choice of solver type, kernel implementation, or computer architecture. The python-nose (<http://code.google.com/p/python-nose/>) unit testing framework simplifies development and selective execution of the tests. Programmatic test code generation is used to exercise the full range of solver and kernel options for each test. Scientific results are archived as application tests within the unit testing framework, ensuring that our published results are reproducible in current and future versions of the PyClaw solver.

In our experience, the application tests are the single greatest factor in facilitating adoption, refactoring, and extension of the code. New users are confident that they have a working installation (or can tell us what doesn’t work on their architectures) and are capable of reproducing our published scientific results. Developers refactoring the code for improved organization, performance, or readability can rely on the application tests for regression testing, to ensure that their changes have not incidentally broken anything. Perhaps most importantly, new solver methods and their implementations can be verified against known solutions with the application tests. This facilitates and encourages the development of new ideas within the PyClaw framework.

**6.2. Hosted Distributed Version Control.** Our use of git (<http://git-scm.com/>), a modern, distributed version control system, provides many benefits. Development need not be synchronized through a master server, which makes it easier to incorporate sub-projects from developers loosely attached to the core team. Management of releases and bugfix updates has been greatly simplified. However, perhaps the greatest beneficiary is the user. Users do not have to wait for PyClaw releases in order to retrieve bugfixes for particular machines or improvements which are under active development, they need only update to a given changeset. Moreover, a user can easily switch between an approved release and experimental code for comparison with a single version control command. This allows the user a much finer-grained manipulation of versioning than was previously

possible.

There are many excellent open source distributed version control hosting sites, including Bitbucket (<http://www.bitbucket.org>) and GitHub (<http://www.github.org>), which provide a range of services to both developers and community users. PyClaw leverages the services provided at GitHub, which includes wiki webpages for user communication, as well as a bug reporting and tracking infrastructure integrated with the hosted version control repository. We have separately engaged the use of Google Groups to provide a mailing list for the PyClaw user and developer community.

**6.3. Documentation.** PyClaw is provided with a range of documentation suitable for the variety of users interacting with the software. While this paper provides a high-level overview of the capabilities of the code and its application, it is our experience from using other projects that the best software documentation includes a separate tutorial and user’s guide with a class reference section. The tutorial and user’s guide are maintained in the ReStructured Text format, from which they can be translated into HTML, PDF, and several other output formats using for instance Sphinx (<http://sphinx.pocoo.org/>). The PyClaw code itself is documented inline using Python’s docstring conventions, allowing us to automatically generate class and module reference sections for our documentation.

**7. Applications.** The numerical algorithms made accessible in PyClaw, empowered by parallelization, are capable of modeling challenging wave propagation phenomena. In this section, we provide three example applications. They are not intended to break new scientific ground, but rather to demonstrate the versatility of the algorithms accessible through PyClaw, and (in the third application) the power of PetClaw as a scalable parallel solver.

**7.1. Shallow Water Flow on the Sphere.** Classical shallow water equations on a sphere are an approximation of the flow on the earth’s surface. They are of interest because they capture most of the flow’s features of a thin layer of fluid tangent to the surface of the sphere. Therefore, they are often used by meteorologists, climatologists and geophysicists to model both atmosphere and oceans.

In three-dimensional Cartesian coordinates, using  $h$  and  $\mathbf{u} = (u, v, w)^T$  to define the height and the fluid velocity vector, the shallow water equations are of the form (2.1), with  $\mathbf{q} = (h, hu, hv, hw)^T$  and

$$\mathbf{f}(\mathbf{q}) = \begin{pmatrix} hu & hv & hw \\ hu^2 + \frac{1}{2}gh & huv & huw \\ huv & hv^2 + \frac{1}{2}gh & hvw \\ huw & hvw & hw^2 + \frac{1}{2}gh \end{pmatrix}, \quad (7.1)$$

where  $g$  is the gravitational acceleration. The source term  $\mathbf{s}(\mathbf{q}, \mathbf{x})$  includes the Coriolis force and an additional term that ensures that the velocity is tangent to the sphere:

$$\mathbf{s}(\mathbf{q}, \mathbf{x}) = -\frac{2\Omega}{a}z(\mathbf{x} \times h\mathbf{u}) + (\mathbf{x} \cdot (\nabla \cdot \tilde{\mathbf{f}}))\mathbf{x}. \quad (7.2)$$

Here  $\Omega$  and  $a$  are the rotation rate and the radius of the earth, respectively. In (7.2)  $\tilde{\mathbf{f}}$  is the part of the flux matrix associated with the momentum equations [2].

In this framework, we consider the numerical solution of the zonal wave number 4 Rossby-Haurwitz problem [30]. Rossby-Haurwitz waves are steadily propagating initial conditions of the nonlinear non-divergent barotropic vorticity equation on a rotating sphere [7]. Although they do not represent exact steady solutions of the shallow water equations, they are expected to evolve in a similar way [29]. For this reason Rossby-Haurwitz waves have also been used to test shallow water numerical models and are among the standard shallow water model test cases proposed by Williamson et al. [30].

The problem consists of a divergence-free initial velocity field that rotates about the  $z$ -axis without significantly changing form on short time scales (dynamical weak instabilities). On longer time scales (50-100 days) the instabilities effects lead to the breakdown of the wave structure. Previous studies have shown that the time at which the solution symmetry breaks depends strongly on the numerical truncation error of the scheme [29]. Because of this the Rossby-Haurwitz problem is frequently used to assess the accuracy of a numerical algorithm for the solution of the shallow water equations on a rotating sphere.

The three-dimensional shallow water equations are solved on the logically rectangular grid introduced in [2], using the same approach employed there, namely the classic Clawpack method with Strang splitting for the source term [28]. Simulations have been performed on four grids with  $100 \times 50$ ,  $200 \times 100$ ,  $400 \times 200$  and  $800 \times 400$  cells. Table 7.1 lists the breakdown time. With the coarsest mesh, the diffusive part of the numerical error suppresses the instability completely (cfr. [29]). The finer grid results confirm that the time at which the initial velocity loses its symmetry is sensitive to the numerical error.

Table 7.1: Time at which the symmetry of the Rossby-Haurwitz wave breaks down.

Grid	Breakdown
$100 \times 50$	-
$200 \times 100$	$\approx 34$ d
$400 \times 200$	$\approx 45$ d
$800 \times 400$	$\approx 46$ d

Figure 7.1 shows the contour line of the water height at day 0 (initial solution), 38, 45 and 48, for the grid with  $400 \times 200$  cells. These plots show qualitatively the evolution of the instability.

**7.2. Shock-Bubble Interaction.** The Euler equations for a compressible, inviscid fluid with cylindrical symmetry can be written as

$$\begin{aligned}
\rho_t + (\rho u)_z + (\rho v)_r &= -\frac{\rho v}{r}, \\
(\rho u)_t + (\rho u^2 + p)_z + (\rho uv)_r &= -\frac{\rho uv}{r}, \\
(\rho v)_t + (\rho uv)_z + (\rho v^2 + p)_r &= -\frac{\rho v^2}{r}, \\
(\rho E)_t + ((\rho E + p)u)_z + ((\rho E + p)v)_r &= -\frac{(\rho E + p)v}{r}.
\end{aligned} \tag{7.3}$$



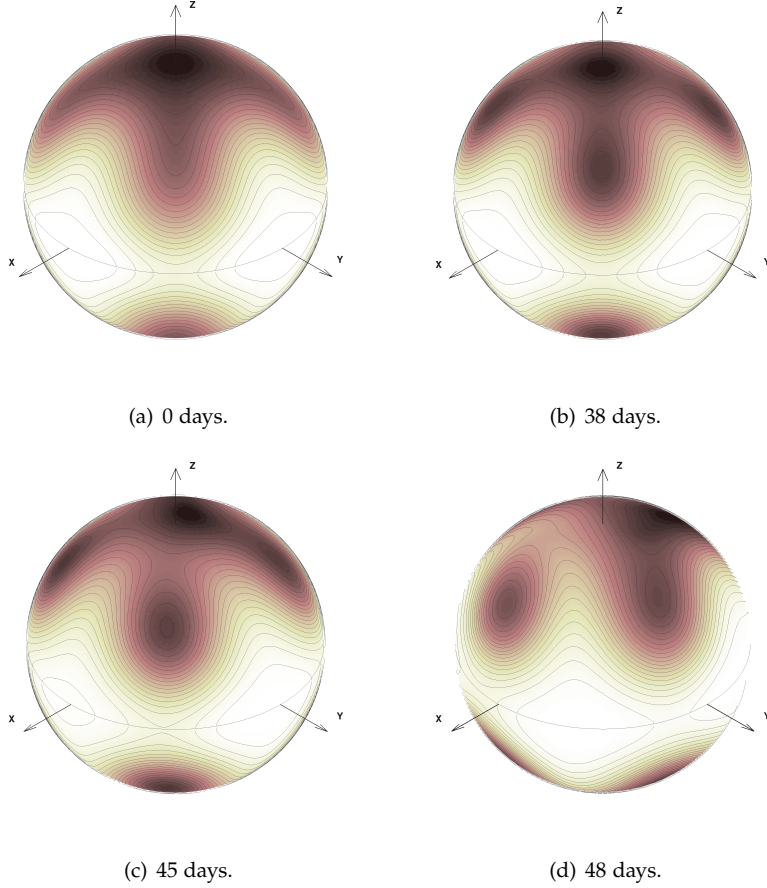


Fig. 7.1: Water height of the zonal wave number 4 Rossby-Haurwitz problem at different times; grid with  $400 \times 200$  cells. Contour interval 120 m.

Here the  $z$ -coordinate represents distance along the axis of symmetry while the  $r$ -coordinate measures distance away from the axis of symmetry. The quantities  $\rho, E, p$  represent density, total energy, and pressure, respectively, while  $u$  and  $v$  are the  $z$ - and  $r$ -components of velocity.

We consider an ideal gas with  $\gamma = 1.4$  in the cylindrical domain  $[0, 2] \times [0, 0.5]$ . The problem consists of a planar shock traveling in the  $z$ -direction that impacts a spherical bubble of lower-density fluid. In front of the shock  $u = v = 0$  and  $\rho = p = 1$  except inside the bubble, where  $p = 1, \rho = 0.1$ . Behind the shock,  $p = 5, \rho \approx 2.82, v \approx 1.61$ , and these conditions are also imposed at the left edge of the domain. In addition to (7.3), we solve a simple advection equation for a tracer that is initially set to unity in the bubble and zero elsewhere in order to visualize where the fluid interior to the bubble is transported. Reflecting boundary conditions are imposed at the bottom of the domain while outflow conditions are imposed at the top and right boundaries.

Since the edge of the bubble is curved, it does not align with the Cartesian grid. Thus, in the cells that are partly inside and partly outside the bubble, the initial condition used is a weighted average of the different solution values, based on the fraction of the cell that is inside. This fraction is computed by adaptive quadrature using the `scipy.integrate.quad` package.

Figure 7.2 shows the initial condition and results of this problem, using a  $1280 \times 320$  grid and the unsplit classic Clawpack algorithm with full transverse corrections. The bubble is observed to transform into a “smoke ring”. Considerably more detailed structure is evident in this simulation compared to lower-resolution adaptively refined results from AMRClaw that are published at <http://depts.washington.edu/clawpack/clawpack-4.3/applications/euler/2d/shockbubble/amr/www/index.html>. Figure 7.3 shows a close-up of the smoke ring solution obtained with the classic Clawpack algorithm, as well as solutions obtained using the SharpClaw algorithm with fifth- (WENO5) and seventh-order (WENO7) reconstructions. All runs were performed on a  $1280 \times 320$  grid with a maximum CFL number of 0.8. Although the overall features of the solutions are similar, more fine structure is apparent in the SharpClaw results. For instance, several vortices can be seen to the left of the smoke ring in the WENO7 run that are not resolved in the classic run.

**7.3. Cylindrical Solitary Waves in a Periodic Medium.** The problem considered in this section is taken from [24]. It involves the propagation of nonlinear waves in a two-dimensional crystal, leading to the formation of solitary radial waves or “rings”. For this purpose, we consider the 2D  $p$ -system with spatially-varying coefficients as a model for elastic waves:

$$\begin{aligned} \epsilon_t - u_x - v_y &= 0, \\ \rho(x, y)u_t - \sigma(\epsilon, x, y)_x &= 0, \\ \rho(x, y)v_t - \sigma(\epsilon, x, y)_y &= 0. \end{aligned} \tag{7.4}$$

Here  $\epsilon$  represents the strain,  $u$  and  $v$  are the velocities in  $x$  and  $y$  respectively,  $\rho(x, y)$  is the spatially-varying material density and  $\sigma(\epsilon, x, y)$  is the stress. The system (7.4) is closed by introducing a nonlinear constitutive relation  $\sigma(\epsilon, x, y)$ . Similar to [17], we take

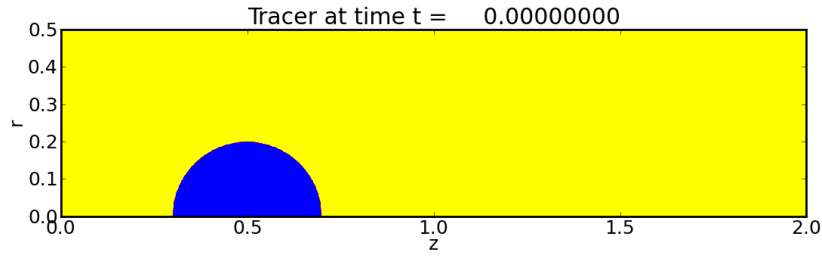
$$\sigma(\epsilon, x, y) = \exp(K(x, y)\epsilon) + 1, \tag{7.5}$$

where  $K(x, y)$  is the spatially-varying bulk modulus.

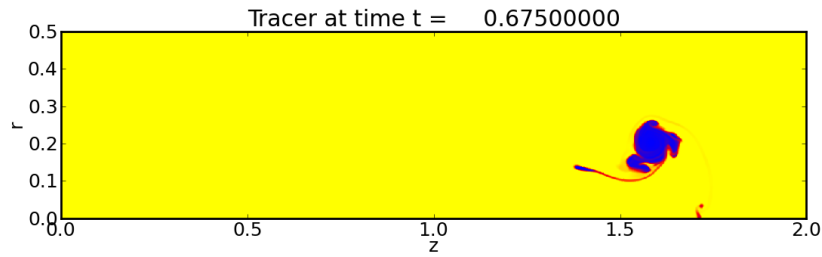
The medium, a small part of which is shown in Figure 7.4(a), is in a checkerboard pattern with alternating squares of two different materials:

$$(K(x, y), \rho(x, y)) = \begin{cases} (1, 1) & \text{if } \left(x - \lfloor x \rfloor - \frac{1}{2}\right) \left(y - \lfloor y \rfloor - \frac{1}{2}\right) < 0 \\ (5, 5) & \text{if } \left(x - \lfloor x \rfloor - \frac{1}{2}\right) \left(y - \lfloor y \rfloor - \frac{1}{2}\right) > 0. \end{cases} \tag{7.6}$$

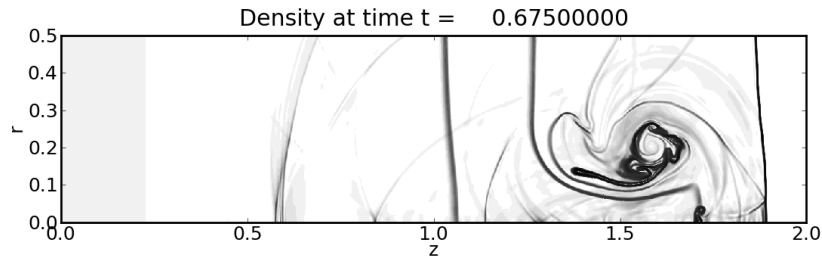
The problem is quite challenging, for multiple reasons. First, the flux is spatially varying and even discontinuous – meaning that the solution variables (strain and momentum)



(a) Initial tracer showing location of low-density bubble.



(b) Tracer showing location of of bubble material after shock impact.

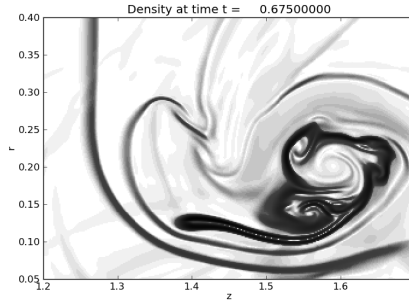


(c) Schlieren plot of density.

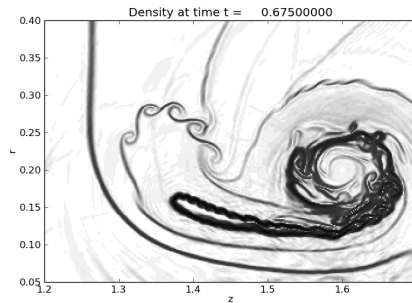
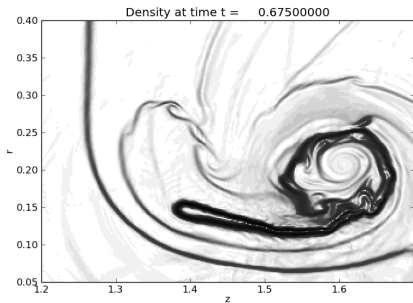
Fig. 7.2: Results of shock-bubble interaction computation, showing the transformation of the initially spherical bubble into a “smoke ring” after it is impacted by a shock wave.

are also discontinuous. Furthermore, these discontinuities include corner singularities. Finally, and owing in part to these discontinuities, it is necessary to use a large number of cells per material period ( $\gtrsim 100$ ) in order to get even qualitatively accurate results. As we are interested in a phenomenon that arises only after waves pass through many ( $> 100$ ) material periods, this leads to a very high computational cost.

The problem is solved using the SharpClaw algorithm with fifth-order WENO reconstruction. As explained in Section 2.1, the implementation in SharpClaw is based on



(a) Classic Clawpack solution.



(b) SharpClaw solution using WENO5 reconstruction. (c) SharpClaw solution using WENO7 reconstruction.

Fig. 7.3: Schlieren plots of density, zoomed in on the smoke ring. All solutions are computed on a  $1280 \times 320$  grid.

solving normal Riemann problems at the grid interfaces; see [25] for a detailed explanation of the approximate Riemann solvers employed and for a much more detailed study of this problem.

A close-up view of the initial stress is shown in Figure 7.4(b). The stress is a Gaussian pulse with an amplitude of 5 and a variance in  $x$  and  $y$  of 5, centered at the origin. The velocity is initially zero. The problem is symmetric with respect to reflection about the  $x$ - and  $y$ - axes, so the computational domain is restricted to the positive quadrant and reflecting boundary conditions are imposed at the left and bottom boundaries. Outflow (zero-order extrapolation) conditions are imposed at the top and right boundaries. In units of the medium period, the domain considered is  $200 \times 200$ , and the grid spacing is  $\Delta x = \Delta y = 1/240$ . Hence the full simulation involves  $6.8 \times 10^9$  unknowns. It was run on 16,384 cores of the Shaheen supercomputer at KAUST, over a period of about 3.2 days.

The formation of solitary wave rings is seen clearly in Figure 7.5(a), which depicts the stress at  $t = 200$ . The structure of these waves is shown in Figure 7.5(b), which displays

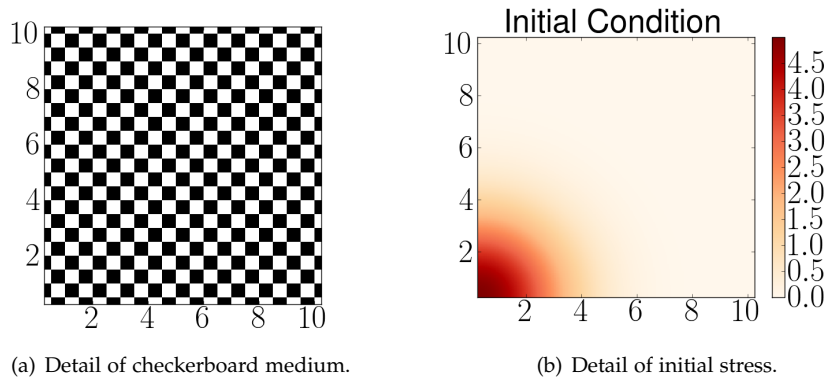


Fig. 7.4: The medium and initial condition (both shown zoomed-in) for the cylindrical solitary wave problem.

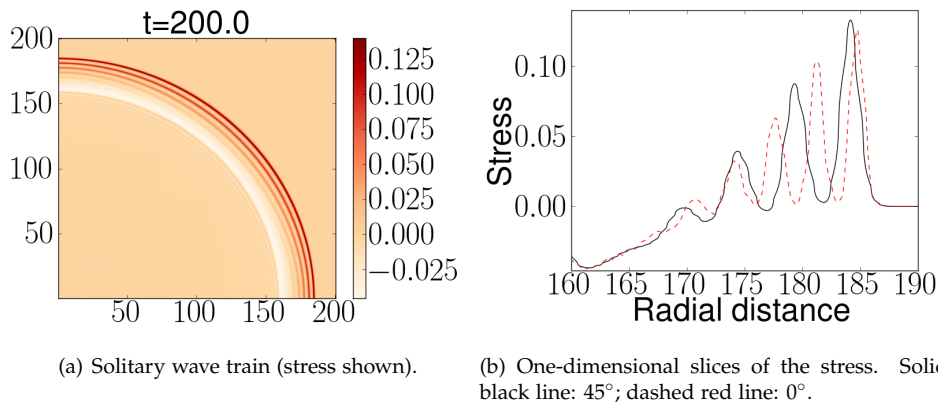


Fig. 7.5: Solution of the  $p$ -system (7.4) in a checkerboard medium, showing 2D cylindrical solitary wave formation.

slices of the stress at  $45^\circ$  (solid line) and  $0^\circ$  (dashed line) with respect to the  $x$ -axis.

**8. Discussion and future plans.** This work demonstrates that the use of Python in combination with existing Fortran and C codes allows the production of scientific software that is accessible, extensible, and efficient. The serial performance loss is relatively small, and is more than compensated for even on a typical laptop by the ability to run in parallel without any additional effort. Combining scalable parallelism with the algorithms of Clawpack and SharpClaw yields a potent tool for exploring novel wave phenomena.

We are in the process of extending PyClaw to include three-dimensional wave propagation and implicit time stepping. Both of these are straightforward steps, since the PyClaw framework was written with such developments in mind and the related software packages (Clawpack and PETSc) already support these features. Preliminary implementations are under testing.

The use of Python in scientific computing has many potential benefits [23]. The Python packages `numpy`, `scipy`, and `matplotlib` offer essential numerical tools with interfaces familiar to users of MATLAB (the *lingua franca* of numerical methods) in a general-purpose programming language. An increasing number of important libraries (like PETSc and Trilinos) now have Python bindings, making it relatively easy to add powerful capabilities like massively scalable parallelism to Python codes. As discussed in Section 6, the Python community promotes a range of positive code development practices that are not common in scientific teams but are often picked up by those who begin to work in Python [6].

While the existence of multiple scientific codes for solving the same problems is healthy, it is widely recognized that increased code sharing and reuse would benefit the numerical analysis and scientific computing communities. Closer integration of code developed by different groups would not only allow researchers to be more productive (by reducing duplication of effort), but would also allow useful algorithmic improvements to be more rapidly distinguished from insignificant ones by simplifying the task of comparing them. In our experience, the adoption of Python as a high-level scientific coding language dramatically increases opportunities for code-sharing and reuse. Indeed, the results described in this paper consist largely of combining a few powerful existing pieces of scientific software.

**Acknowledgments.** We are very grateful to Randall LeVeque, and all the Clawpack developers, without whom PyClaw would not exist. The authors also thank the following people:

- Christiane Helzel, for providing Fortran code used in the shallow water example
- Jed Brown and Lisandro Dalcin, for help in improving serial performance
- Hans Petter Langtangen, for encouragement and for advice on drafts of this work

MGK acknowledges partial support from DOE Contract DE-AC01-06CH11357. We thank the KAUST supercomputing lab for allocation of time on the Shaheen supercomputer. Finally, we thank the referees for many helpful comments that led to improvements of this paper.

## REFERENCES

- [1] X. CAI, H. P. LANGTANGEN, AND H. MOE, *On the performance of the Python programming language for serial and parallel scientific computations*, *Scientific Programming*, 13 (2005), pp. 31–56.
- [2] D. A. CALHOUN, C. HELZEL, AND R. J. LEVEQUE, *Logically rectangular finite volume grids and methods for circular and spherical domains*, *SIAM Review*, 50 (2008), pp. 723–752.
- [3] M. EMMETT, *PyWENO software package*, 2011. <http://memmett.github.com/PyWENO>.
- [4] J. ENKOVAARA, N. A. ROMERO, S. SHENDE, AND J. J. MORTENSEN, *GPAW - massively parallel electronic structure calculations with Python-based software*, *Procedia Computer Science*, 4 (2011), pp. 17–25.
- [5] S. FOMEL AND J. F. CLAERBOUT, *Guest Editors' Introduction: Reproducible Research*, *Computing in Science and Engineering*, 11 (2009), pp. 5–7.
- [6] J. E. GUYER, D. WHEELER, AND J. A. WARREN, *FiPy: partial differential equations with Python*, *Computing in Science and Engineering*, 11 (2009), pp. 6–15.
- [7] B. HAURWITZ, *The motion of atmospheric disturbances on a spherical earth*, *Journal of Marine Research*, 3 (1940), pp. 254–267.
- [8] D. I. KETCHESON, *Highly efficient strong stability preserving Runge-Kutta methods with low-storage implementations*, *SIAM Journal on Scientific Computing*, 30 (2008), pp. 2113–2136.

- [9] D. I. KETCHESON AND R. J. LEVEQUE, *WENOCLAW: A higher order wave propagation method*, in *Hyperbolic Problems: Theory, Numerics, Applications: Proceedings of the Eleventh International Conference on Hyperbolic Problems*, Berlin, 2008, p. 1123.
- [10] D. I. KETCHESON AND M. PARSANI, *SharpClaw software*, 2011. <http://numerics.kaust.edu.sa/sharpclaw>.
- [11] D. I. KETCHESON, M. PARSANI, AND R. J. LEVEQUE, *High-order wave propagation algorithms for general hyperbolic systems*. Submitted, 2011.
- [12] H. P. LANGTANGEN AND X. CAI, *On the efficiency of Python for high-performance computing: A case study involving stencil updates for partial differential equations*, in *Modeling, Simulation and Optimization of Complex Processes*, H. G. Block, E. Kostina, H. X. Phu, and R. Rannacher, eds., Springer, 2008, pp. 337–358.
- [13] P. LAX AND X. LIU, *Solution of two-dimensional Riemann problems of gas dynamics by positive schemes*, *SIAM Journal on Scientific Computing*, 19 (1998), pp. 319–340.
- [14] R. J. LEVEQUE, *Wave propagation algorithms for multidimensional hyperbolic systems*, *Journal of Computational Physics*, 131 (1997), pp. 327–353.
- [15] R. J. LEVEQUE, *Finite volume methods for hyperbolic problems*, Cambridge University Press, Cambridge, 2002.
- [16] R. J. LEVEQUE AND M. J. BERGER, *Clawpack Software version 4.5*, 2011. <http://www.clawpack.org>.
- [17] R. J. LEVEQUE AND D. H. YONG, *Solitary waves in layered nonlinear media*, *SIAM Journal of Applied Mathematics*, 63 (2003), pp. 1539–1560.
- [18] K. T. MANDLI, D. I. KETCHESON, ET AL., *PyClaw software*, 2011. Version 1.0.
- [19] K. A. MARDAL, O. SKAVHAUG, G. T. LINES, G. A. STAFF, AND A. ØDEGÅRD, *Using Python to solve partial differential equations*, *Computing in Science and Engineering*, 9 (2007), pp. 48–51.
- [20] J. MORTENSEN, L. HANSEN, AND K. JACOBSEN, *Real-space grid implementation of the projector augmented wave method*, *Physical Review B*, 71 (2005).
- [21] J. NILSEN, X. CAI, B. R. HØYLAND, AND H. P. LANGTANGEN, *Simplifying the parallelization of scientific codes by a function-centric approach in Python*, *Computational Science & Discovery*, 3 (2010), p. 015003.
- [22] T. OLIPHANT, *NumPy Web page*, 2011. <http://numpy.scipy.org>.
- [23] F. PEREZ, B. E. GRANGER, AND J. D. HUNTER, *Python: An ecosystem for scientific computing*, *Computing in Science and Engineering*, 13 (2011), pp. 13–21.
- [24] M. QUEZADA DE LUNA, *Nonlinear Wave Propagation and Solitary Wave Formation in Two-Dimensional Heterogeneous Media*, MSc thesis, King Abdullah University of Science and Technology (KAUST), 2011. <http://numerics.kaust.edu.sa/papers/quezada-thesis/quezada-thesis.html>.
- [25] M. QUEZADA DE LUNA AND D. I. KETCHESON, *Radial solitary waves in two-dimensional periodic media*. in preparation., 2011.
- [26] J. SHI, C. HU, AND C.-W. SHU, *A technique of treating negative weights in WENO schemes*, *Journal of Computational Physics*, 175 (2002), pp. 108–127.
- [27] V. STODDEN, M. G. KNEPLEY, C. WIGGINS, R. J. LEVEQUE, D. DONOHO, S. FOMEL, M. P. FRIEDLANDER, M. GERSTEIN, I. MITCHELL, L. L. OUELLETTE, N. W. BRAMBLE, P. O. BROWN, V. CAREY, L. DENARDIS, R. GENTLEMAN, D. GEZELTER, J. A. GOODMAN, J. E. MOORE, F. A. PASQUALE, J. ROLNICK, M. SERINGHAUS, AND R. SUBRAMANIAN, *Reproducible Research: addressing the need for data and code sharing in computational science*, *Computing in Science and Engineering*, 12 (2010), pp. 8–13.
- [28] G. STRANG, *On the construction and comparison of difference schemes*, *SIAM Journal on Numerical Analysis*, 5 (1968).
- [29] J. THUBURN AND L. YONG, *Numerical simulations of Rossby-Haurwitz waves*, *Tellus A*, 52A (2000), pp. 181–189.
- [30] D. L. WILLIAMSON, J. B. DRAKE, J. J. HACK, R. JAKOB, AND P. N. SWARZTRAUBER, *A standard test set for numerical approximations to the shallow water equations on the sphere*, *Journal of Computational Physics*, (1992), pp. 221–224.